chap 3

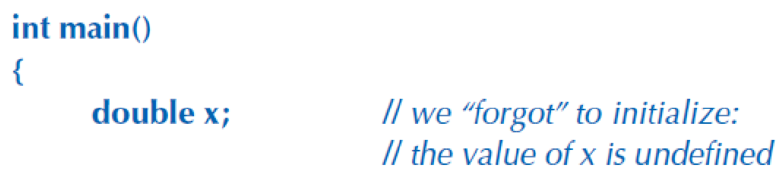
how many bits/bytes each data type takes

Can use `sizeof variable` to check how many bytes it has.

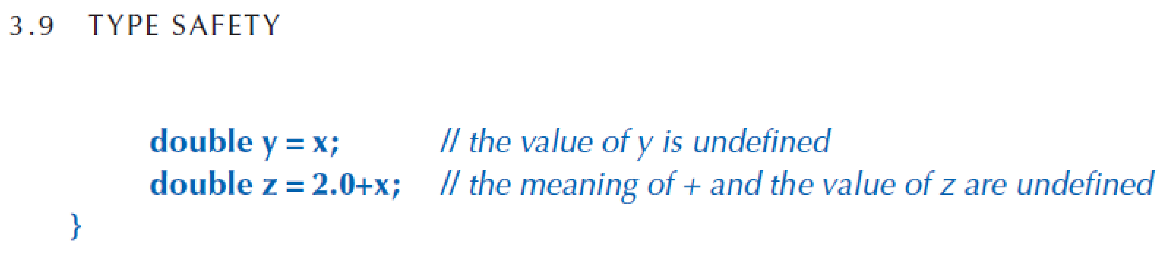
- String each character in string takes 1 byte. But string ALWAYS end with a null character, so that's an extra byte. "\n" is a newline character and counts as just 1 byte, because only 1 character.
- char always 1 byte, does not end with null character
- int is 4 bytes = 32bits
- bools and chars each 1 byte and is fixed
- double is 8 bytes and fixed
- Note that size of types is **implementation-defined**, meaning that it CAN vary among different machines. Once again, use sizeof function if not sure what size a type is.

c++ uninitialized variables strange behaviors

In general never work with uninitialized variables as to trusting them to be some kind of value. If you want to be safe always initialize them to something. If you are 100% sure that you are only going to assign a value to it later on, then it might be ok. Here's an example that is not okay



```
int main()
{
        double x;             // we "forgot" to initialize:
                              // the value of x is undefined
```



3.9 TYPE SAFETY

```
        double y = x;         // the value of y is undefined
        double z = 2.0+x;     // the meaning of + and the value of z are undefined
}
```

Safe conversions

char -> int -> double.

```
char c = 'x'
int i1 = c
double i2 = i1
```

The ascii character value for lowercase x is 120, so i1 is 120 and i2 is 120.0.

other safe conversions

bool to char
bool to int
bool to double
char to int
char to double
int to double

For math, never use floating points, the roundings will fuck up the computations, I experienced that on euler problem 64 with continued fractions. The solution was to either use fraction class or decimal class. The decimal class has custom precision, you can set it yourself, to have more precision than floating point and int. In financial calculations it's important to avoid float and instead use something with better precision.

unsafe conversions

double to int
double to char
double to bool
int to char

int to bool
char to bool

Unsafe in the sense that the original value might not end up being the same after the conversion.

- double -> int -> will truncate, take away whatever decimal part was. Just tested it, it will always round down.
- double -> bool -> everything that is not zero, negative or positive, will be true. If it is zero, then false.
- int -> char -> same as double->char and char is signed because int can be negative and positive.
- int -> bool -> same as double, zero gives false and everything else true
- char -> bool -> any character that is not the null character \0 will be true, otherwise false.

string -> int

Will convert to zero.

double -> char

double -> char, will truncate the decimal part and only use the integer part, which will correspond to the ascii character of the integer. If it goes over the valid values for ascii, which is usually [-128,127] on PC for signed chars, then something involved happens. It will wrap around the value by using modulo 256 (1 char is 1 bytes = 8 bits = 2^8 = 256 possible values) and depending on if it's unsigned [0,255] or signed [-128,127], it will give different answers. Let's assume the input was 1000.5 as double and we want to convert to char. This will happen in both cases

$$1000 \mod 256 = 232$$

Now one of two things will happen depending on if unsigned char or signed char. On my PC it's signed char, which I think is for most PCs.

1. signed char

   $$232 - 255 = -24$$

   But -24 is not an ascii value, so it will output a non-printable control character.

   However if the number had been negative then instead it would have been 24, because 232 is remainder when it's positive, so we be at -232, but the procedure for negative values is to add 256 until it's positive then stop, so $-232 + 256 = 256 - 232 = 24$.
2. Unsigned char
   Since we got 232 we are done, nothing more to do here, but once again 232 is

not an ascii value, so output non-printable control character.

Signed char for negative numbers
If the number is negative we get

$$-1000 \mod 256 = 256 - (1000 \mod 256) = 24$$

The idea is:
When taking the mod of a positive number we will never go over the number we are taking mod on, always be under it, in this case always under 1000. So if the remainder for the positive is 232, then we are missing 232 to reach the original number. But since we are dealing with a negative value -1000, that means we need at least 232 to reach 0 (going in the other direction, think of it as the 1D number line), so we add 256 again to make it positive giving $-232 + 256 = 256 - 232 = 24$.

```
-1000 -----------------0-------256-----------1000
                   ^                       ^
                   |                       |
    (-232 missing to reach 0)    (232 missing to reach 1000)
```

Unsigned char negative numbers
We get 24 apparently the same idea? Add 256 until you get a positive value.

Why keep unsafe conversions?
For historical reasons, C had it. In reality for experienced and most situations narrowing conversions don't pose issues if you know what you are doing. Compilers warns about narrowing conversions.

Universal and uniform initialization
C++11 introduced a convention for initialization to mitigate at least weird initialization values creeping into programs. Surround initial value with braces instead of assigning with =

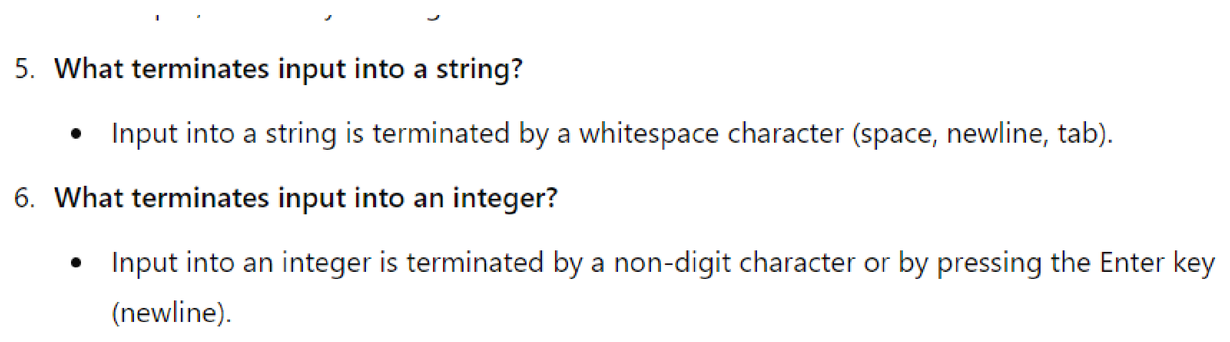
```
double x {2.7};
int y {x};
char b1 {1000}; --> gives narrowing error by compiler
```

This is called universal and uniform initialization.

Small note on char range of values

Char on pc $[-128, 127]$, but only $[0, 127]$ can be used portably, since not every computer is PC, different computers have different ranges $[0, 255]$.

Terminating characters I/O



5. **What terminates input into a string?**
   - Input into a string is terminated by a whitespace character (space, newline, tab).
6. **What terminates input into an integer?**
   - Input into an integer is terminated by a non-digit character or by pressing the Enter key (newline).

object definition = region of memory with a type that specifies what kind of information can be placed in it.

Types in bits

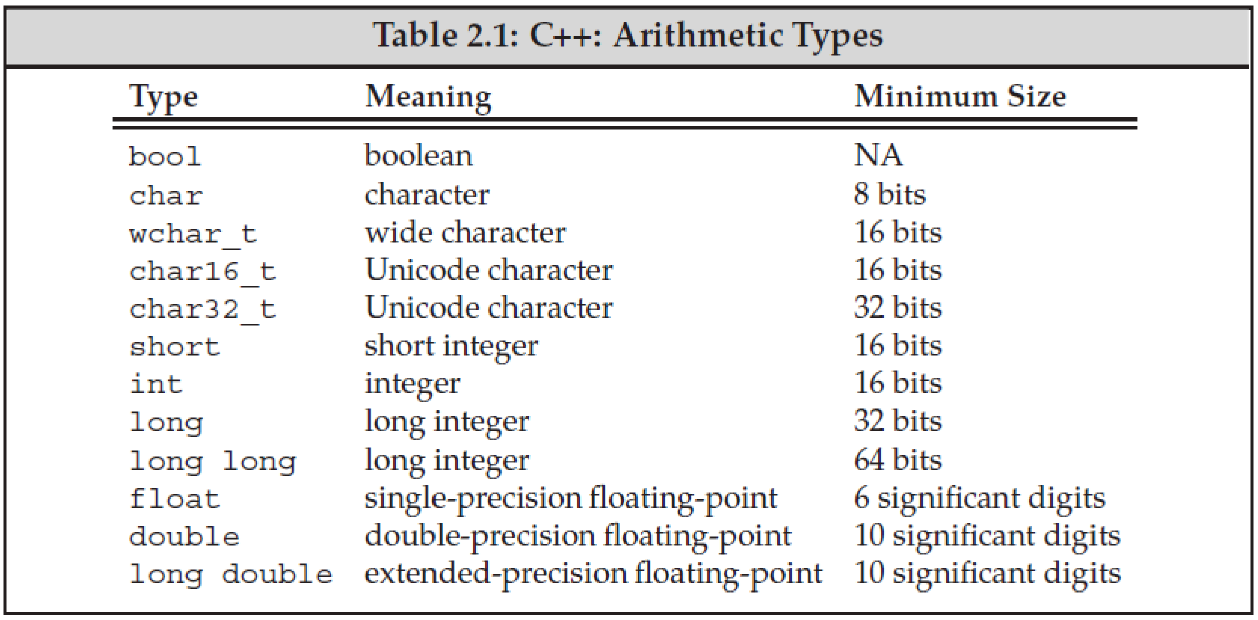
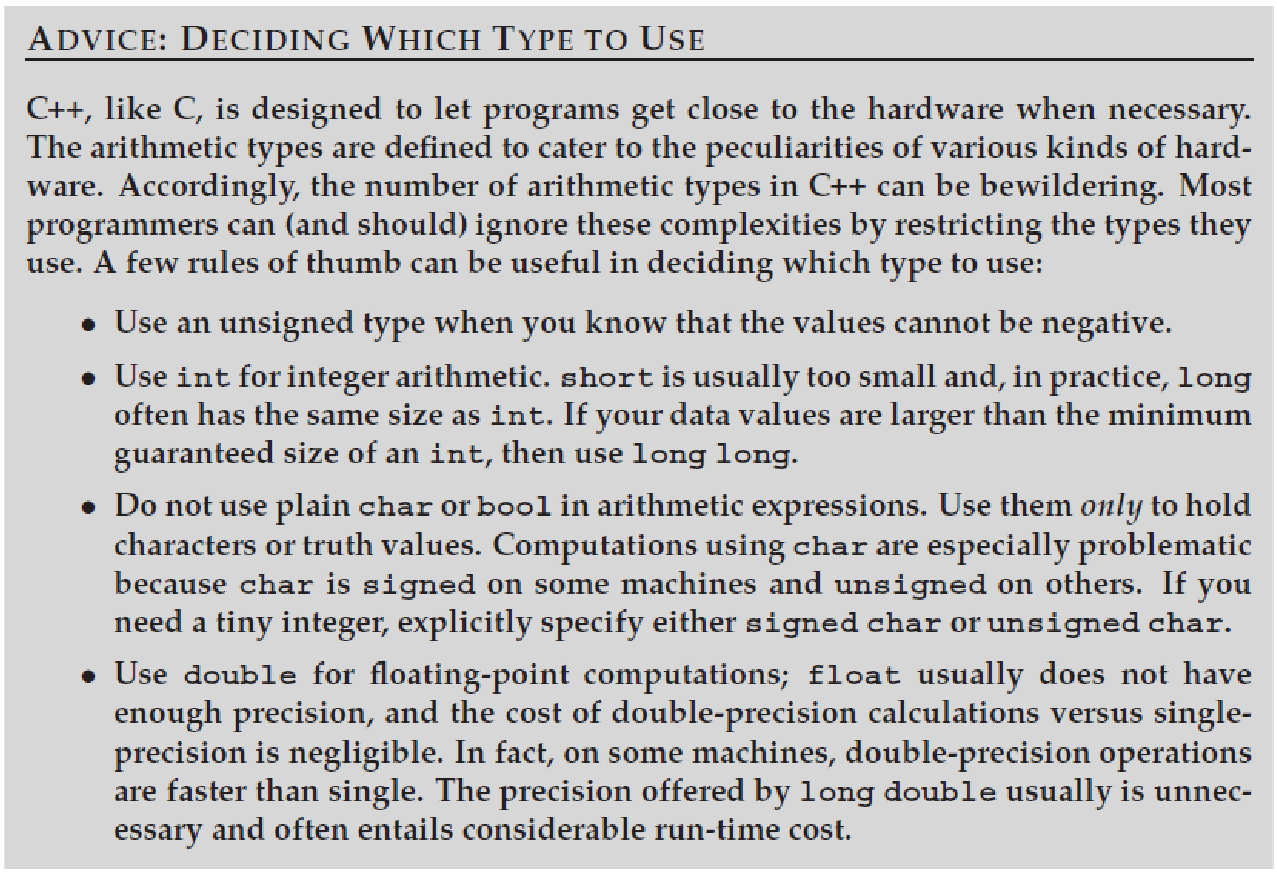


Table 2.1: C++: Arithmetic Types

Type	Meaning	Minimum Size
`bool`	boolean	NA
`char`	character	8 bits
`wchar_t`	wide character	16 bits
`char16_t`	Unicode character	16 bits
`char32_t`	Unicode character	32 bits
`short`	short integer	16 bits
`int`	integer	16 bits
`long`	long integer	32 bits
`long long`	long integer	64 bits
`float`	single-precision floating-point	6 significant digits
`double`	double-precision floating-point	10 significant digits
`long double`	extended-precision floating-point	10 significant digits

Advice on choosing types in c++

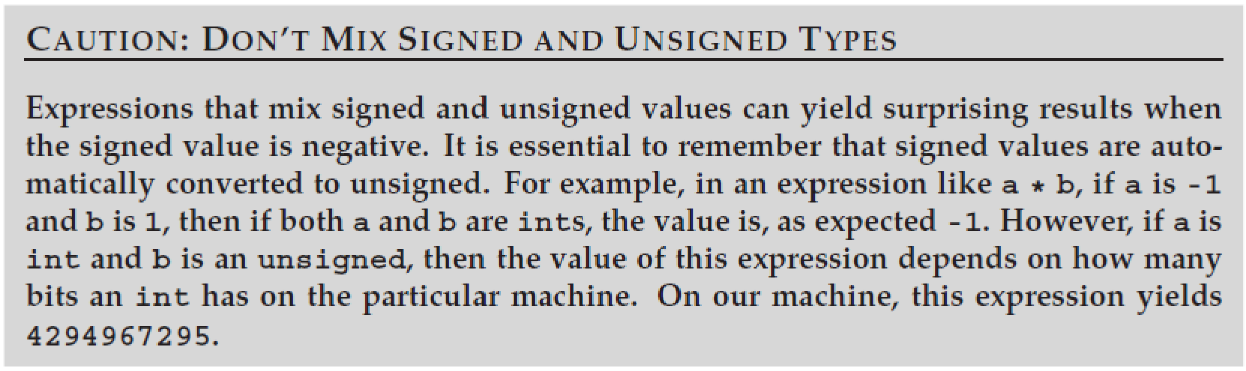


ADVICE: DECIDING WHICH TYPE TO USE

C++, like C, is designed to let programs get close to the hardware when necessary. The arithmetic types are defined to cater to the peculiarities of various kinds of hardware. Accordingly, the number of arithmetic types in C++ can be bewildering. Most programmers can (and should) ignore these complexities by restricting the types they use. A few rules of thumb can be useful in deciding which type to use:

- Use an unsigned type when you know that the values cannot be negative.
- Use `int` for integer arithmetic. `short` is usually too small and, in practice, `long` often has the same size as `int`. If your data values are larger than the minimum guaranteed size of an `int`, then use `long long`.
- Do not use plain `char` or `bool` in arithmetic expressions. Use them *only* to hold characters or truth values. Computations using `char` are especially problematic because `char` is `signed` on some machines and `unsigned` on others. If you need a tiny integer, explicitly specify either `signed char` or `unsigned char`.
- Use `double` for floating-point computations; `float` usually does not have enough precision, and the cost of double-precision calculations versus single-precision is negligible. In fact, on some machines, double-precision operations are faster than single. The precision offered by `long double` usually is unnecessary and often entails considerable run-time cost.

don't mix unsigned with signed



CAUTION: DON'T MIX SIGNED AND UNSIGNED TYPES

Expressions that mix signed and unsigned values can yield surprising results when the signed value is negative. It is essential to remember that signed values are automatically converted to unsigned. For example, in an expression like `a * b`, if `a` is `-1` and `b` is `1`, then if both `a` and `b` are `int`s, the value is, as expected `-1`. However, if `a` is `int` and `b` is an `unsigned`, then the value of this expression depends on how many bits an `int` has on the particular machine. On our machine, this expression yields `4294967295`.

general notes on chap 3

- Whitespace acts as terminator for cin.
- **cin truncating string input for int or double variable**
  `cin >> <numerical variable like int or double>` will truncate inputs that starts with a number but is followed by anything non-numeric like "2asödjk9i2daokpdksada", will truncate it to "2" and convert it to a number. So this is bad if some validation is needed but not implemented. Can give unexpected behavior of a program that doesn't handle these cases. The simplest way I found was to use `std::stringstream` from `<sstream>` library.

- **cin.fail() - check if type conversion was successful or if reading input in general was successful**
  `cin >> a`, but a is for instance int and user provided non-numeric string "adjas", then it will try to convert string to int, which will give 0. To check if it failed to properly convert can call `cin.fail()`.
  - If you assign an out-of-range value to a type then the behavior is **undefined**. The program might appear to work, crash or produce garbage value.
  - Should avoid implementation-defined behavior for types, since the limit of types can vary between different machines. Such programs are said to be nonportable.
  - ***Beware of int used for division when need double**
    `9/5` will give 1 because numerator and denominator are ints. One way to solve this is to make one of them double.
    `9.0/5`
  - ***using for loop with cin instead of while -> skips defining local variable for cin**

```
for (Entr y ee; cin>>ee; ) // read from cin into ee
cout << ee << '\n'; // wr ite ee to cout
```

- **Compiling files with -o flag**
  Do not do this `g++ -o test.cpp test`, it will override the cpp file with empty file. The right way to do it is `g++ test.cpp -o test`, so the cpp file should come first before the -o flag.
- Use sstream if want to automatically handle user input were some kind of number is expected. It does alot of heavylifting parsing the input for you. And for `+<number>` and `-<number>` it will interpret them as positive and negative respectively.
-